Venice: Exploring Server Architectures for Effective Resource Sharing

Jianbo Dong[1], Rui Hou[1], Michael Huang[2], Tao Jiang[1], Boyan Zhao[1], Sally A. McKee[3], Haibin Wang[4], Xiaosong Cui[4], and Lixin Zhang[1]

[1]SKL Computer Architecture, ICT, CAS, *{dongjianbo, hourui, jiangtao, zhaoboyan, zhanglixin}@ict.ac.cn*

[2]University of Rochester, *michael.huang@rochester.edu*

[3]Chalmers University of Technology, *mckee@chalmers.se*

[4]Huawei Technologies Co., Ltd, *{benjamin.wanghaibin, cuixiaosong}@huawei.com*

ABSTRACT

Consolidated server racks are quickly becoming the backbone of IT infrastructure for science, engineering, and business, alike. These servers are still largely built and organized as when they were distributed, individual entities. Given that many fields increasingly rely on analytics of huge datasets, it makes sense to support flexible resource utilization across servers to improve cost-effectiveness and performance. We introduce Venice, a family of data-center server architectures that builds a strong communication substrate as a first-class resource for server chips. Venice provides a diverse set of resource-joining mechanisms that enables user programs to efficiently leverage non-local resources.

To better understand the implications of design decisions about system support for resource sharing we have constructed a hardware prototype that allows us to more accurately measure end-to-end performance of at-scale applications and to explore tradeoffs among performance, power, and resource-sharing transparency. We present results from our initial studies analyzing these tradeoffs when sharing memory, accelerators, or NICs. We find that it is particularly important to reduce or hide latency, that data-sharing access patterns should match the features of the communication channels employed, and that inter-channel collaboration can be exploited for better performance.

1. INTRODUCTION

Users and organizations increasingly rely on large-scale datasets in science, engineering, and business. With growing datasets, modern workloads increasingly involve the the cooperation of many servers simultaneously. Consolidated data centers are thus rapidly becoming the main computing platforms, yet their architecture has remained relatively static, consisting largely of a simple aggregation of commodity servers. There thus remain several mismatches between modern workloads and conventional server architectures, primarily in the communication substrate.

Server networking is primarily concerned with reaching remote destinations. As a result, nodes are typically connected in a tree topology. Even communicating with a neighboring node requires traversing long cables and going through massive, energy-intensive switches with non-trivial latencies. First, studies have shown that, unlike in a campus environment where most traffic (95%) is outbound (north-south), in a data center environment, the majority of the traffic is internal (east-west) [1]. Second, conventional networking interfaces are designed for environments with long, often unreliable connection media. Error handling and other protocol overheads coupled with relatively slow hardware interfaces may not create performance issues for campus environments, but they become more problematic with increasing server-to-server communication and interaction.

In this traditional organization, servers can be viewed as little more than isolated islands, much like the 118 individual islands located in the marshy Venetian lagoon. Only by connecting the islands with bridges do we get the city of Venice that we know today. Moving forward, we are seeing less rigid boundaries among servers within a data-center server rack. Many resources may need to be flexibly shared in order to make such servers more versatile and cost-effective than their isolated ancestors.

Here we propose the Venice (Figure 1) family of experimental architectures. Venice supports highly transparent, multimodal communication for flexibly sharing resources among nodes. Note that other proposals and designs exist for systems that can use remote resources [2, 3, 4]. For instance, some systems support RDMA [3], but performance for random, fine-grain access is insufficient to support direct sharing of remote memory. As another example, Scale-out NUMA [2] supports remote memory usage but requires programs to be rewritten using a new programming model, as opposed to just linking with a new library.

Based on our analyses we believe Venice requires the symbiosis of three things:

978-1-4673-9211-2/16/$31.00

1. **Integration of communication:** To minimize latencies, the interconnect fabric needs to be directly integrated on-chip rather than accessed via I/O buses and adapters.

2. **Multi-modality:** To translate raw fabric capabilities into application performance benefits, the architecture and software stack must support common communication patterns for resource sharing. These include explicit communication and remote memory accesses at both coarse and fine granularities.

3. **Transparency:** The system needs to provide interfaces that not only minimize user-level runtime overheads but require few, if any, changes to user programs.

The designs for other proposed systems that share remote resources [2, 3] are often evaluated with simulations that are subject to biases introduced by the assumptions and abstractions that we necessarily build into our modeling tools. To better understand some of the design issues and their implications, we built an experimental prototype that faithfully implements remote accesses within a cluster.

Accessing remote resources will always be less efficient than accessing local ones, and sharing beyond some distance may not be profitable. Even with the support Venice provides, the resources within a collection of servers cannot appear completely "flat". Venice makes resources appear flatter, though, by utilizing them more efficiently. It represents a first step in exploring the design space of effective server architectures, but several important questions remain to be addressed in greater depth, e.g., fault containment and scalability. Nonetheless, this paper makes three main contributions to the design of data-center server architectures:

- We propose a novel architectural design;
- We construct and study a fully functional hardware prototype — an instance of a Venice architecture — to expose technical issues and provide faithful evaluations of long-term system behavior; and
- We discuss design rationales, experimental results, and some lessons learned to instill a more concrete understanding of the opportunities, limitations, and challenges of resource sharing in server environments.

Our experiments show that fully utilizing communication channels requires that the granularity of communication (e.g., page, block, or cacheline transfers) match the fabric capabilities. In particular, low latency is crucial for fine-grained communication, and thus hiding and/or reducing latencies is necessary for efficient resource sharing. Finally, when a node integrates multiple communication channels, inter-channel collaboration delivers the best performance.

2. RELATED WORK

The most common resource-sharing examples are symmetric multiprocessor (SMP) and hardware distributed shared-memory (DSM) systems that target scientific and high-performance computing (HPC). Such systems deliver high performance, but they also come with high price tags [5, 6, 7]. More cost-effective software DSM systems incur significant overhead maintaining cache coherence [8, 9]. Like us, other researchers have studied leveraging remote idle memory for swap space [10, 11, 12], checkpoint storage [13], or file caching [14, 15], but these solutions retain fabrics intended for HPC communication, not data center traffic. Note that resource utilization in systems targeting HPC is notoriously low, which drives the TCO unacceptably high for clouds.

Many higher-end HPC systems employ specialized interconnects to support efficient communication. For instance, the Cray T3E uses a custom ASIC to instantiate a 3D torus network that provides much higher bandwidth, faster chip interfaces, and better scalability than previous systems [16]. Other examples include the IBM PERCS switchless interconnect [17], the Tofu interconnect for the Fujitsu K supercomputer [18], and the TianHe-1A interconnection fabric [19]. These expensive, custom solutions are intended to support MPI, sockets, and PGAS language-based HPC applications having very different traffic patterns from data center software. Their first-order design goals are performance and reliability, not low power, high density, incremental scaling, and flexible resource sharing.

In contrast, AMD SeaMicro's Freedom Fabric [20], Calxeda's Fleet Fabric [21], and Intel's Rack-Scale Architecture [22] specifically target cloud platforms. These high-density computing fabrics disaggregate the compute, memory, and storage capacities within a rack to support more flexible resource sharing and scaling, but the latter two sacrifice software transparency by implementing custom APIs. Intel RSA introduces high-speed, fat bandwidth (off-chip) silicon photonic links [22]. This architecture also supports logically composing physically separated pooled resources. Finally, PCIe has also been explored as a low-cost, flexible, but lower-performing data center interconnect [23, 24, 25].

Researchers have also begun pursuing promising approaches to remote resource sharing in data center platforms. For instance, Lim et al. [26, 27, 28] introduce disaggregated memory, which allows multiple compute blades to access dedicated memory blades via HyperTransport or PCIe interconnects. Novakovic et al. [2] propose Scale-out NUMA (so-NUMA), an architecture, programming model, and communication protocol for using remote memory. They use an RDMA-inspired programming model and a stateless messaging protocol on top of a NUMA memory fabric. Putnam et al. [29] implement Catapult, which accesses distributed accelerators through an FPGA-based network. All these projects focus on sharing one specific remote resource, whereas Venice provides a common substrate to enable experimentation with more holistic approaches to sharing multiple types of resources. Note that so-NUMA and disaggregated memory have thus far been explored only in simulation, which necessarily abstracts away many low-level details of hardware implementation. In contrast, we provide a hardware vehicle to facilitate more realistic design-space exploration. The current prototype is thus a strawman experimental system, not an end-product.

3. THE VISION FOR VENICE

To better explain the composition and operation of our proposed Venice architectures, we start with a concrete ex-

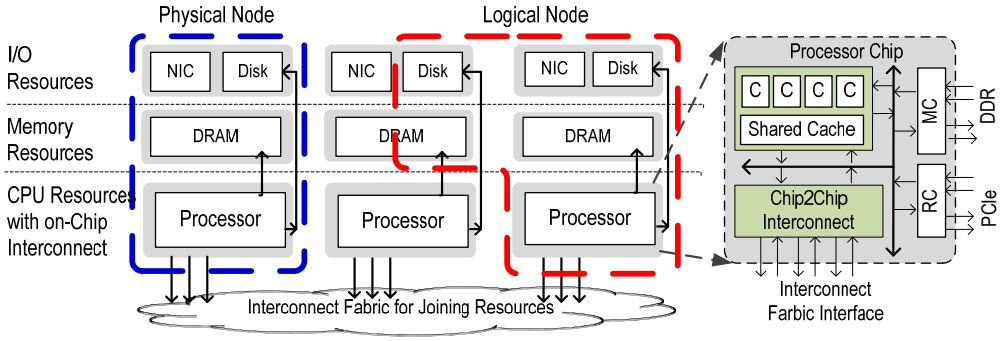


Figure 1: Venice architecture organization

ample of a likely memory-sharing scenario. We then discuss the high-level structure of Venice support for such sharing.

Consider a node running an in-memory database like Redis. In a conventional server, each node has its own fixed set of resources, including memory. The DB application can find out the memory size, and it may execute system calls to request *all* of it. Venice systems report memory availability to software running on a Monitor Node (MN), much like a NameNode in a Hadoop system (① in Figure 2). When a node requests more memory than is locally available, the kernel memory manager sends a request to the MN (②). The MN[1] finds an appropriate donor node, tells the donor's software agent to hot-remove a portion of its memory, sets up a Venice interface to service requests to that region (③), and tells the recipient node to hot-plug the newly acquired memory region and to set up its own Venice interface (④). Once this connection is established, accesses to the borrowed region on the recipient node are intercepted by hardware and routed to the donor node.

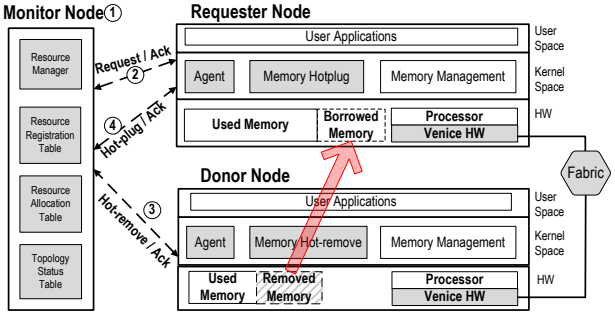


Figure 2: Memory sharing flow diagram

The ability to access remote resources exists in many current systems — e.g., modern systems allow remote memory access via RDMA over a fast network [30]. But such capabilities are not designed with true resource sharing in mind and do not perform well enough to use remote resources as if they were local. At the other end of the spectrum, tightly coupled shared memory architectures have the most flexibility for resource sharing. Shared-memory systems let nodes share the same resource *simultaneously* and provide fine-grained flexibility, but maintaining cache coherence and consistency models incurs large performance and energy costs. This makes large-scale shared-memory systems too expensive to deploy in cost-conscious data centers.

Venice architectures will break the traditional boundaries of physical nodes (Figure 1). Each Venice node can easily (transparently to user-level applications) and efficiently (with tolerable latencies) utilize idle remote resources. Recall that multiple requesters only use the same resource via time sharing. To enable such resource sharing, our prototype Venice system embodies three layers of support.

Resource sharing fabrics: Historically, conventional networks were non-central system components with long connections over unreliable media and deep software stacks, which makes them a poor match for datacenter traffic. And although PCIe provides a straightforwad interface by which to connect nodes, but it was designed for peripheral management, and current implementations perform poorly for fine-grained communication. In contrast, Venice systems are more sensitive to fabric performance, especially latency. To reduce communication overheads, our prototype (1) integrates the fabric directly on chip, avoiding speed bumps in the I/O buses, adapters, and converters; (2) employs an ultra-lightweight protocol intended for short-distance links in reliable, well controlled data center environments; and (3) avoids relying on large, distant switches (as in Ethernet or IB) that further contribute to performance overheads for common-case usage of proximal resource sharing, e.g., within a rack.

Mapping and joining mechanisms: Raw interconnect performance is useless if user-level software must traverse thick layers of middleware to access remote resources. Venice architectures thus accelerate common tasks in hardware. Venice supports a set of efficient resource-joining mechanisms that maintain user-level transparency. For example, accessing remote memory via CRMA (through load/store instructions) requires no special API, and the software abstraction layer removes the need for applications to be aware of accelerator location.

Runtime management: The previous two layers provide the mechanisms to use remote resources. Venice runtime systems then manage the allocation and release of those resources. An agent on each node periodically collects resource usage information and reports it to the central MN. When a node needs resources beyond its local capacity, it

[1]Note that the MN must be spared to avoid a single point of failure; our small prototype does not yet implement this redundancy.

sends a request to the MN, which selects appropriate donor nodes. We describe our prototype's current, skeletal runtime implementation in the next section, but broader analysis of the runtime design space is beyond the scope of this architectural discussion.

With these three enabling components, our Venice architectures can support at least two kinds of resource sharing: borrowing resources in independent nodes and accessing dedicated, "passive" resource pools, such as memory or IO boards. These capabilities allow single nodes to access larger scales of resources and enable more powerful and flexible resource provisioning within cloud architectures.

4. LATENCY ANALYSIS

Whether future servers can make efficient use of remote resources depends largely on the performance of the interconnect between participating nodes. Therefore we first perform a feasibility study to discover whether the latencies of today's high-performance interconnects (together with their software stacks) are low enough to support extensive cross-node resource sharing.

4.1 Limitations of Commodity Interconnects

We start with a legacy system of x86 servers connected by commodity interconnects such as 10Gb Ethernet and InfiniBand, based on which we build a set of software solutions to access remote memory[2]. We also build a semi-custom PCIe interconnect solution that supports direct remote memory access either via on-demand cacheline fills (Cacheline Remote Memory Access, or CRMA) or via swapping over the block device using DMAs. For all configurations, we provide 4GB local memory and execute the BerkeleyDB with a 6GB array and a random access pattern. The read-write ratio is set to 80-20, which is typical for OLTP databases. Fig. 3 shows execution times of the sharing configurations normalized to that of using only local memory.

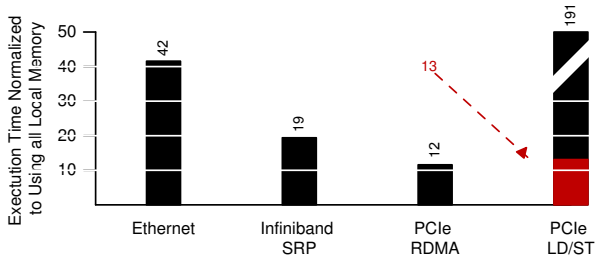


Figure 3: Remote memory efficiency with commodity interconnects. Note that the PCIe CRMA configuration suffers from a crippling, but fixable, limit due to the commodity PCIe chip.

This comparison shows that accessing remote memory over commodity interconnects experiences varying degrees of slowdown compared to having all local memory. This is due to the demanding access pattern stressing the interconnect — other, more forgiving applications or access patterns may tolerate the overheads of commodity interconnects better. Nevertheless, it is clear that commodity interconnects are not (yet) designed to support fast, random remote access: they incur overheads from multiple sources. Relying only on them to support inter-node resource sharing will greatly limit its efficiency, and thus its practicality.

[2]For 10Gb Ethernet, we use remote memory as a swap partition via a vDisk driver in Linux. We use a similar setup for InfiniBand, leveraging its SCSI RDMA Protocol (SRP) to generate the virtual block device.

4.2 Impact of Architectural Support

We next investigate the potential benefits of adding architectural support for remote resource access. We employ a second prototype platform (shown in Fig. 4) that consists of an array of Xilinx ZC706 boards [31], each equipped with an FPGA and ARM Cortex-A9 processors. The nodes run a complete software stack including the Linux OS (Linaro 13.09). We implement the interconnect fabrics in programmable logic, connecting the ARM processors via AXI buses with support for cache coherence.

Figure 4: Eight-node prototype connected by a 3D-mesh network

To accurately estimate the performance of target systems whose bottlenecks potentially differ from our prototype's, we use the programmable logic to allow us to set throughput caps and to insert delays to slow down relatively fast components. This gives us much more modeling acuity than simulation-based studies. For instance, when validating our prototype against an Intel Xeon E5620 server running the same workloads and software stack, the wall-clock times we measure are consistently about $1/16^{th}$ those on the target machine (within 10% variation).

4.2.1 *Impact of latency tolerance/reduction*

With this prototype we can emulate designs with more built-in support for reducing remote access latency. We contrast these designs with those that try to *tolerate* such latencies. In particular, we implement the design proposed in Scale-out NUMA, which uses a user-level asynchronous programming style over a Queue Pair (QPair) communication channel. Such latency tolerance works best with applications like PageRank that present few data dependencies among transactions.

Figure 5 compares the performance of several systems running PageRank and BerkeleyDB[3]. We start with a legacy

[3]For BerkeleyDB, we set up the client with 1000 transactions com-

configuration (labeled "off-chip QPair" in the figure) in which remote data is accessed through a QPair channel on an IB network. Next, we implement QPair support mechanisms on-chip (labeled "on-chip QPair") to reduce access latencies. In the third configuration, we rewrite the application to orchestrate the software-based asynchronous communication proposed in Scale-out NUMA [2]. In the next two configurations (labeled "off-chip CRMA" and "on-chip CRMA"), we add hardware to support cacheline fills from a remote node with off-chip and on-chip interface logic, respectively. For these configurations, the code needs no explicit communication, but instead it accesses remote memory as if it were local: once the connection is set up, cache misses to remote memory are automatically handled by the hardware. In all cases, the application uses a gigabyte of data in the memory of a remote node that is directly connected (i.e., without an intermediate router node). We normalize execution time of all configurations to a system having all memory local.

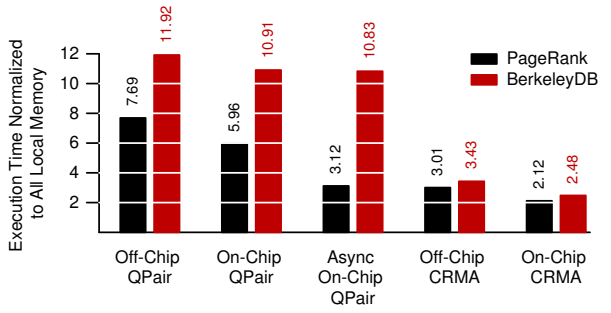


Figure 5: Relative performance of our system configurations

Our experiments show that when the algorithm permits, redesigning the communication pattern to hide latencies can deliver significant performance gains. PageRank's massive parallelism can be exploited to initiate multiple streams of communication in the background, thereby tolerating remote access latencies. In this case, performance is comparable to building off-chip support for cacheline fills from remote memory. When the cacheline fill support is integrated on-chip, we see another performance boost (1.4x).

For BerkeleyDB, the asynchronous QPair shows very few performance benefits over legacy QPair. This is because the client must check the return status before processing the next query.

Based on these results, we conclude that hardware support for remote memory access is clearly worthwhile. First, the benefit is real. Even for applications with significant parallelism that can be exploited by a sophisticated software implementation, the performance with hardware support for memory access is still higher — with no extra application programmer effort.

Second, the support need not be complex. Note that remote memory access does not necessitate hardware cache coherence. We envision a "single-subscriber" model in which the OS/hypervisor of a physical node ensures that a region of memory is owned by a single node at any time.

posed of five random queries (four gets and one put). The server stores the records in remote memory. When a query arrives, the key is used to look up the address of the corresponding record.

The hardware support then amounts to address translation and packetization. In fact, the hardware cost of such support is less than that of on-chip QPair. A typical QPair implementation supports hundreds of queue pairs, each requiring around a dozen registers. This requires tens of kilobytes more SRAM than does CRMA. And the logic complexity (in terms of LUT counts) of QPair is about twice that of CRMA.

4.2.2 *Impact of router delay*

In the experiments above, we directly connect the two nodes via an optical link. Our measurements suggest that the latency of the physical layer (PHY) is a significant, and sometimes dominant, component of overall transaction latency. This means that when the interconnection between two resource-sharing nodes relies on an indirect network with an external router (which is typical for datacenters), the additional hop should noticeably increase end-to-end latency. To understand the impact on performance, we repeat the experiment with a router inserted between the two nodes. Figure 6 shows performance results for this configuration.

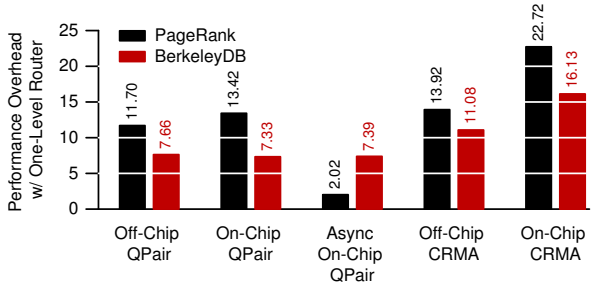


Figure 6: Performance impact of off-chip router delay

Obviously, the impact of additional router delay is greater for higher-performing configurations. The only exception is when the code already hides latency well via asynchronous programming. For configurations supporting CRMA, the impact of going through an external router is large (over 20%). Given this, direct chip-to-chip communication will be a desirable feature for direct remote memory accesses.

To recap, our end-to-end performance evaluations (of real hardware) show:

1. Commodity interconnects, especially those with relatively thick protocol stacks, still limit performance of extensive cross-node resource sharing. Using remote resources over commodity interconnect is an order of magnitude slower than using local resources.

2. Additional architectural support for high-performance communication can be very effective, bringing remote-access penalties down to much more tolerable levels (e.g., 2-3×).

3. For remote resource usage to be highly effective, latency reduction is crucial. Application-level latency tolerance is effective for some workloads, but not all.

4. In our opinion, reducing latency requires on-chip support for remote access, and it probably requires support for direct interconnections.

These observations motivated a number of elements in the design of Venice that reduces access latency of remote resources. We describe our design next.

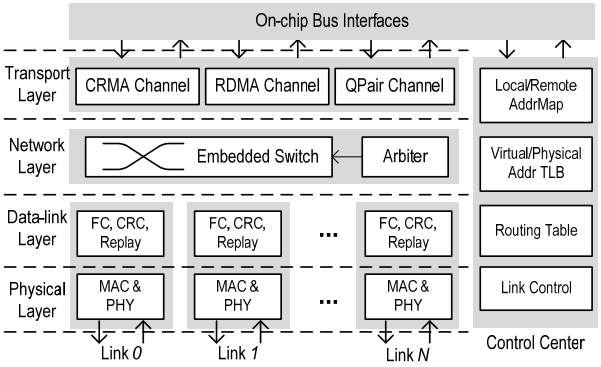


Figure 7: Fabric structure

5. OUR VENICE DESIGN

Recall that Venice represents a family of architectures, allowing much flexibility in individual implementations. Here we describe the details of the initial implementation the we designed as a vehicle for design-space exploration.

5.1 Resource-sharing fabrics

The interconnect forms the foundation of Venice architectures. Its performance — in terms of both latency and bandwidth — directly impacts the effectiveness of resource sharing. We therefore integrate the Venice interconnect directly into the processor to avoid the overhead of going through additional chip interfaces, I/O hubs, and/or adapters. We embed the switch to allow direct chip-to-chip communication without relying on intermediary switch modules. Section 7.3 discusses the costs of such integration. Of course, external switch modules are still important for scalability.

5.1.1 Fabric structure

Figure 7 shows the structures of the IC and protocol. The Venice prototype supports basic flow control and error correction. It can also support emerging features such as flow-based QoS for software-defined networks, but these are beyond the scope of this discussion. The aforementioned functionalities are implemented in its *Control Center* and the transport, network, datalink, and physical layers. We discuss these layers bottom-up.

Datalink and physical layers. The datalink and physical layers support multiple I/O ports. The datalink is responsible for point-to-point reliable transmission. We use credit-based flow control to prevent buffer overflow at the receiver. Error detection with Cyclic Redundancy Check (CRC) on the receiver side and a corresponding replay mechanism on the sender side guarantee packet correctness.

Network layer. A main design decision was to make the fabric capable of operating in a "switchless" mode for direct chip-to-chip communication. By not traversing faraway central switch modules, this mode provides low-latency and low-power communication to neighboring nodes. The benefits justify the reasonable hardware implementation costs with respect to pin requirements, on-chip real-estate, power, and design complexity.

We believe the on-chip switch will be of low dimension: first, the chip will be pin-limited, which makes it difficult to support namy wide channels, and second, we expect Venice to be most cost-effective when sharing resources within a small diameter. For more remote sharing, an external, high-dimension switch module can complement the on-chip switch.

5.1.2 Remote access channels

To ensure that raw fabric performance is not masked by an inefficient software stack, we support multiple mechanisms for giving user-level applications direct access to remote resources. Venice's transport layer mechanisms consist of a CRMA channel, an RDMA channel, and a QPair channel. As suggested by the name, the light-weight CRMA channel supports remote memory accesses via direct load/store instructions. The RDMA channel is more suitable for large-volume data movement. And the QPair channel supports user-level, socket-based communication. Each channel has its own address window in local memory, and thus Venice implements a *Remote Address Mapping Table* (RAMT) and a *Transport-Layer TLB* (TLTLB) (Figure 7) to facilitate address translation.

CRMA channel. Remote memory access in Venice is set up by software on both the local (recipient) and remote (donor) nodes. The local node first allocates address space for remote memory blocks. It then sends the donor node a memory-sharing request specifying the amount of memory needed. After the handshake with the remote node, it creates an entry in the RAMT (Figure 8). The remote node creates a matching entry in its own mapping table. When the processor issues a memory request to an address in the RAMT, the CRMA channel captures, packetizes, and sends it to the donor. Finally, either of the nodes may initiate a stop-sharing request to its counterpart. Relevant entries from the RAMTs in both nodes are invalidated after proper cleanup.

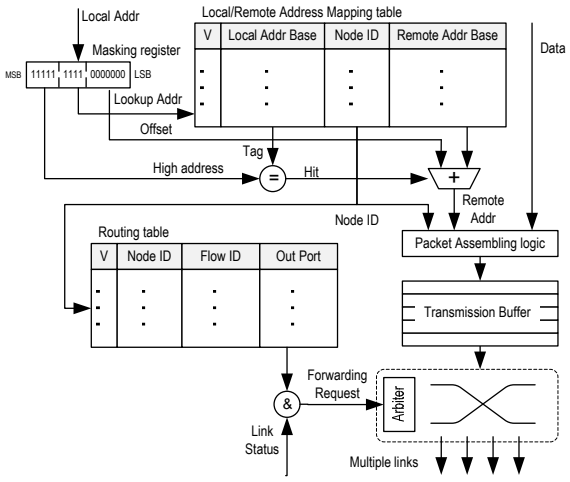


Figure 8: Remote address mapping/routing tables

RDMA channel. Whereas the CRMA channel serves cacheline requests (fetches or writebacks) triggered by in-

dividual memory instructions, the RDMA channel handles software-initiated DMA requests with remote memory as the source/destination. State machines and control registers divide the memory region into chunks for packetization.

QPair channel. Venice's QPair mechanism is a bidirectional channel between two communicating threads. Once established, data written into the local send queue will be delivered to the counterpart's receive queue. The benefit of the QPair is that the well defined, low-level queue management maps well to hardware state machines. This frees up the CPU and transfers efficiently large blocks.

5.1.3 *Inter-channel collaboration*

Based on our experience with the prototype, we find all three channels to be necessary, even though any one channel can mimic the functionality of others. In particular, the CRMA channel is most efficient when the access pattern is random (as for in-memory databases) or the communication granularity is small. We therefore implement an adaptive communication library that makes intelligent decisions about channel choices based on communication demands and that allows channels to supplement each other. This means that the channels are no longer independent entities: packets may arrive out of order, necessitating a sequence number (something we learned the hard way).

Consider credit-based flow control, which is widely used in communication systems like the Sockets Direct Protocol (SDP) over InfiniBand. Credits indicate the number of available buffers at the receiver. Before each transmission, the sender checks the credit and only continues the transfer (and decrements the credit) when there are enough buffers available. In a traditional design, flow-control packets are transmitted through a QPair channel just like data packets. However, the relatively long latency for these small packets brings down link bandwidth utilization. We thus propose to send flow-control packets through the CRMA channel to reduce the latency for updating credits, which we store in a separate memory region. These flow-control packets are overwriteable and irrelevant to data packets, allowing us to simplify the collaboration between the channels.

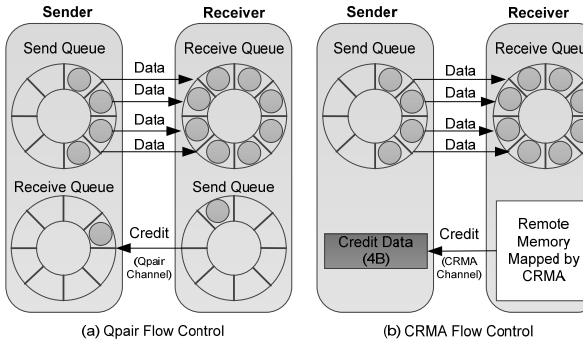


Figure 9: Credit packet transmission via QPair and CRM

5.2 Resource-sharing mechanisms

Venice is comprised of several kinds of hardware sources, including computing resources (CPUs, GPGPUs, and accelerators), memory, disk, and other IO devices. Programming models like Hadoop, MPI, sockets, or PGAS languages provide good support for using remote CPU and disk resources (e.g., via NFS or NAS). Using remote memory, accelerators, and other IO devices is more challenging, since their use often imposes stringent requirements on the performance of the sharing mechanisms. These challenges are exactly what Venice architectures are intended to address.

We provide software mechanisms that closely mesh with the hardware to complement the Venice architecture. These mechanisms follow three general design guidelines:

- To enhance programmer productivity, we keep resource joining as software-transparent as possible. Applications are concerned only with allocating the resources they need, without regard to their whereabouts.
- To maximize portability, we leverage existing (Linux) interfaces to enable remote resource access.
- To maximize efficiency, the software and hardware are tightly coupled and highly optimized.

5.2.1 *Using remote memory*

Nodes report idle memory regions to the MN, after which remote nodes may request them. On such a request, a region is removed from the control of the local OS and managed henceforth by the recipient node. The functionality of removing a memory region from the view of the software is already supported by Linux.

Direct remote memory access. Perhaps the most user-transparent use of remote memory is to treat it as ordinary local memory. To do so, we need to present the remote region to the (local) OS, for which we use the Linux memory hot-plug mechanism. Figure 10 illustrates this process. Initially (step 0), both Node A and Node B contain 4GB physical memory, of which 1GB (high addresses in the figure) in Node A is reported as available to share. In step 1 of allocating this memory to Node B, the region is hot-removed and no longer visible to software on Node A. In step 2, Node B uses hot plugging to create the illusion of additional memory. The Venice hardware on both nodes is configured so that accesses to the new address region on Node B (0x1 0000

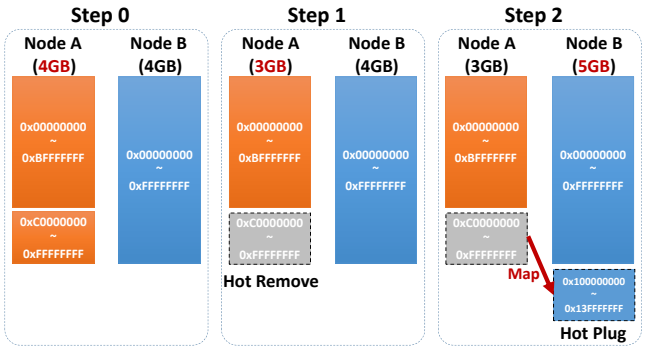


Figure 10: Direct remote memory access

Remote memory as swap space. We implement a high-performance virtual block device to interact with remote

memory used for page swapping. To reduce interrupt overheads, our device driver uses double buffering to interact with descriptors for RDMA transactions. The driver can use memory regions from multiple nodes by presenting them as multiple block devices.

5.2.2 *Using remote accelerators*

Sharing accelerators can improve utilization and reduce over-provisioning in a data center environment. To maintain transparency for user applications, Venice abstracts accelerators as message-passing mailboxes (implemented as buffers pinned in memory), as shown in Figure 11. A mailbox contains: (1) a request buffer for storing executables to run on the accelerator, (2) an input data buffer, (3) a return data buffer, (4) a task start flag, and (5) a completion flag. A kernel thread running on the donor node processes the mailbox and launches tasks on remote accelerators on behalf of recipient nodes.

Venice provides an optimized communication path for donor accelerators that are exclusively shared with one recipient. The accelerator access interface (memory-mapped buffers and control registers) is exclusively mapped to the recipient node similarly to how a memory region is shared. The recipient directly manipulates the accelerator input and output buffers, which improves efficiency on both nodes.

As a concrete example, when an application needs accelerators, it uses our API to invoke library calls that request accelerator(s) from the resource management middleware (running on the monitor node). The management middleware makes information about each allocated accelerator — including node ID and mailbox base address — available to the library.

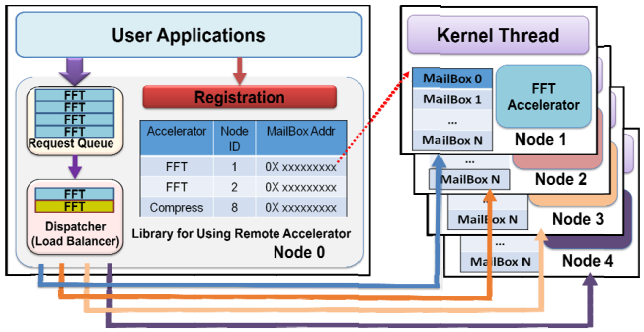


Figure 11: Using remote accelerators

Figure 11 shows an example in which an application on Node 0 receives two FFT and one crypto accelerator. Accelerator details are abstracted away from the application, which merely sends requests through the library. The library handles all details, including dispatching tasks using the right channel to send to each accelerator mailbox.

5.2.3 *Using remote NICs*

Venice supports dynamically leveraging remote NICs to increase network bandwidth for network-bound applications (or even application phases). Figure 12 shows our IP-over-QPair interface (labeled VNIC in the figure) that enables TCP/IP traffic to seamlessly traverse the QPair channel. Specifically, we have developed a pair of drivers (front-end and back-end) to emulate the NIC. The drivers share a MAC address. If Node 0 wants to use the NIC on Node 1, the Node 0 front-end driver presents the NIC interface to applications, and the Node 1 back-end driver forwards packets to the real Node 1 NIC via the software network bridge. The Linux network bonding mechanism combines the local and emulated NICs on Node 0 to create a single, virtual network interface. One hardware QPair services each IP-over-QPair connection.

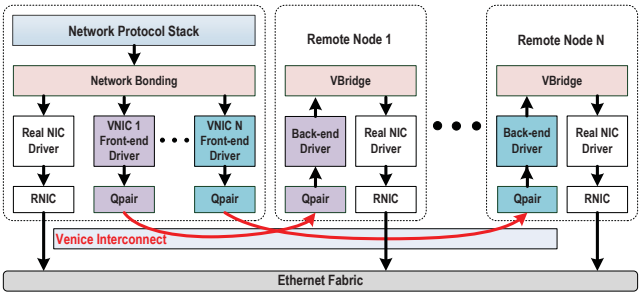


Figure 12: Using remote NICs

5.3 Resource-management runtime

Recall that Venice systems include three layers. The design of the third, the runtime system, is a broad topic involving interactions among reliability, scalability, QoS, and various policy choices in resource allocation and optimization. Exploring this rich design space falls outside the architectural discussion presented here, and much of it belongs to future work. Our Venice prototype will facilitate that exploration.

To set up sharing, we implement a skeletal version of the management runtime, which we describe here. The global view of available resources, their allocation status, and the status of system components (e.g., links) is maintained by three tables in the MN.

1. The *Resource Registration Table* (RRT) tracks available resources in the rack. The RRT contains metadata for each resource, including address, size, and capabilities. A daemon process in each node collects availability information and periodically reports to the MN, serving as a heartbeat for the MN to infer node status.

2. The *Resource Allocation Table* (RAT) tracks all allocation records. The RRT and RAT give the MN a global view of available resources.

3. The *Topology Status Table* (TST) tracks fabric link status. The daemon tests and reports the status of the Venice fabric links on every heartbeat.

With this information, the MN responds to requests for remote resources by allocating from the most appropriate donor nodes. The allocator should consider distance between potential donor and recipient, the nature of the sharing (and thus bandwidth demand), and existing traffic over involved links. Given the scale of our prototype, our current algorithm only considers distance. Note that it is possible

for MN records to be stale, allowing it to ask for more idle memory than are currently available. We employ handshake and retry mechanisms to address this.

6. EXPERIMENTAL METHODOLOGY

Our experimental analysis primarily relies on our hardware prototype, as shown in Fig. 4. With the prototype system, we can expose hidden technical issues that will inform the design of subsequent Venice systems. In contrast, a conventional simulation-only approach only models performance effects (rather than actually carrying out actions), and thus inherently ignores unforeseen interactions (and fails to shed light on the unavoidable uncertainties in interactions among the many parameters). In addition, the prototype allows us to observe long-term behavior in production-scale application scenarios. Indeed, comparing our simulation results from sampled execution [32] with those from complete simulations (costing thousands of hours) reveals discrepancies of up to 6.6× in performance gains. Prototype measurements agree with full-length simulations much more closely, with the remaining difference probably being due to the OS (our simulations are user-mode only).

Table 1 shows the configurations for our evaluations. Our workloads include Hadoop [33] and Spark [34] applications from BigDataBench [35], Graph500 [36], and an in-memory database [37]. We use FFT [38] and Iperf [39] to study specific components in greater detail.

Table 1: Platform configuration

Hardware Parameters	
System	8 nodes, 3D mesh
Nodes	Xilinx ZC706, Linux (Linaro 13.09)
Processor	ARM Cortex-A9, 667MHz
Memory	1GB SODIMM (active)
Fabric	Parallel/serial clock 125MHz/5GHz, P2P latency 1.4μs, bandwidth 5Gbps×6
Big Data Applications	
Hadoop	Grep, 9.7GB Dataset
Spark	Connected Components (CC), 8192 nodes, 21461 edges
Graph500	R-MAT scale 22, R-MAT edge factor 14
Inmem DB	Redis w/ 1-2.5GB memory Berkeley DB w/ 1GB dataset
MySQL	400M entries×64B dataset
PageRank	vertices 1488712, edges 8678566
Other Benchmarks	
SPLASH2	FFT 512MB input
IPerf	4-256B packets

7. EVALUATION RESULTS

We perform quantitative experiments to better understand resource sharing on Venice. Specifically, we conduct three case studies of sharing different types of remote resources. We then analyze several architectural design decisions. Finally we discuss hardware cost.

7.1 Resource-sharing case studies

Leveraging remote memory. We construct a mini data-center system to run a (near) real-world workload consisting of a database query service running in parallel with several instances of Spark's Connected Components (CC). Figure 13 shows the six-node system, which includes two x86 servers and four nodes of our Venice prototype.

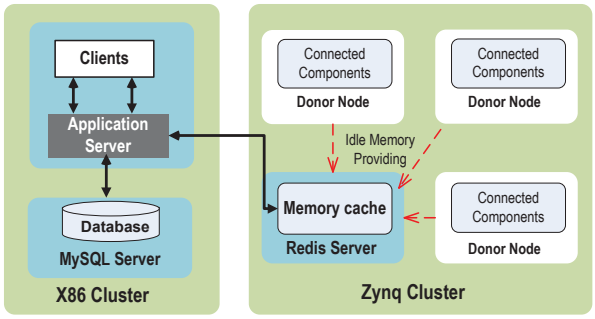
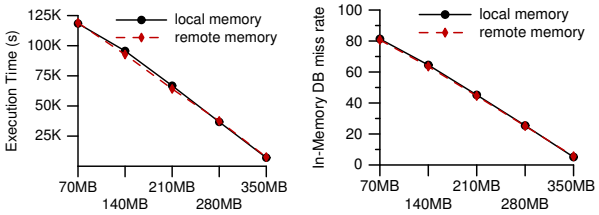


Figure 13: Our mini data-center system

One x86 server runs MySQL [40], and one of the Venice nodes runs the Redis [41] in-memory database service that works as key/value cache. The MySQL database and the Redis server collaborate to create a typical web-service architecture. When the application server receives a query from the clients, it first checks the Redis server for a match. Only upon a miss will the database server be accessed. The other x86 server works as an internet client issuing online query requests, dispatches queries, and collects results. The remaining Venice nodes run CC graph analyses, which mimics a situation where nodes with CPU-intensive workloads can contribute other resources such as memory.

To see the effect of Redis server using remote memory, we keep only a minimum amount of local memory (50MB) for Redis to start properly. The rest of the memory is provided by donor nodes. We measure a few configurations with different amount of remote memory (in 70MB increments). All execution time is measured after propoer initialization and warmup. Figure 14 shows performance for 10000 random queries as we vary the total amount of memory.



(a) Execution Time (b) Miss Rate

Figure 14: System performance

As we can see, the Redis hit rate clearly improves with increases in memory resources, which is the major reason of performance improvement. The corresponding execution times decrease from 11900s (with 70MB memory) to 758s (with 350MB), a 15.7x performance improvement. When contrasting the performance using remote and local memory, we can see that there is very slight difference, because the time spent on missed queries dominates the overall execution time. When the miss rate decreases to near 5% (the last set of results), the benefit of having memory being local

(7% faster) is more visible. We note that the performance impact to the workload running on doner nodes (CC) is negligible because the memory traffic caused by remote sharing is insignificant.

Figure 15 compares performance of accessing remote memory either directly (through the CRMA channel) or as swap space via the RDMA channel. For reference, we also show performance when local memory is big enough (ideal), or when the portion of remote memory is supplied by swapping to local storage as is the conventional case.

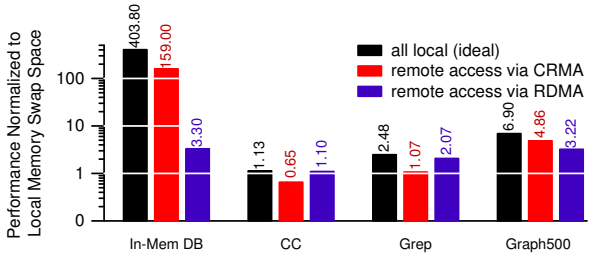
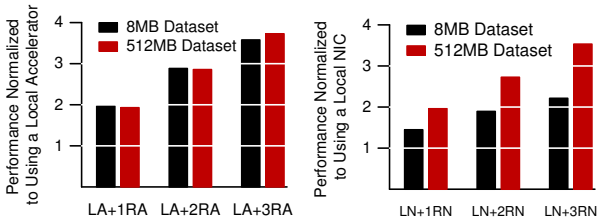


Figure 15: Remote memory access performance with 75% remote memory and 25% local memory. All results are normalized to the case where local swapping is needed to supply memory capacity.

We make three observations. First, memory is a critical resource. If swapping (to local storage) is avoided by adding sufficient local memory, performance can be orders of magnitude higher. Second, with the high-efficiency support of Venice, remote memory can be quite effective. Relative to using all local memory, the slowdown is limited to 1.03x to 2.5x. Compared to the costly resource overprovisioning, sharing nearby resource can be much more cost-effective. Finally, applications have different access locality: some favor direct fine-grain access to remote memory, while others benefit more from page-level swapping. The difference in speed between the two modes is non-trivial (up to 6.8×). This shows the utility of supporting different modes.

Leveraging remote accelerators. Hardware accelerators that execute special (offloaded) computations are common resources in modern systems. Most of these devices are designed to only accelerate portions of typical workloads. As such, some are likely to be most efficient as shared resources. To analyze the impact of the Venice communication substrate when sharing acclerators we implement SPLASH2 [38] FFT on a Xilinx board (XFFT) and compare the effects of using the accelerators locally or remotely through Venice. Figure 16a summarizes performance results normalized to those for using a local accelerator. Performance improves almost linearly with the number of accelerators, indicating that Venice introduces insignificant overheads throughout the entire system stack.

Leveraging remote NICs. Figure 16b shows performance for sharing remote I/O devices. We use the iPerf toolset to measure bandwidth when using remote NICs. For clarity, we focus on two representative configurations (sending tiny, 4B packets and larger, 256B packets) and only show observed throughput. Venice overhead is noticeable for the tiny packets. With three remote NICs, effective utilization of available bandwidth is around 40%. Utilization



(a) Remote Accelerators (b) Remote NICs

Figure 16: Performance benefits of sharing remote accelerators and NICs. LA and RA (LN and RN) stand for local and remote accelerators (NICs), respectively.

for more "normal" 256B packes quickly rises to about 85%. This again confirms that when sharing nearby resources over Venice, the system imposes insignificant overhead for all but the most fine-grain communication patterns.

Note that all these studies are limited by the scale of our prototype. Studying the effects of sharing faraway resources or sharing multiple resources that may cross paths with one another is part of future work, as is studying the effects of queuing delay from potential bottlenecks (such as the Monitor Node).

7.2 Analysis of multi-modality

Venice supports three transport channels each of which can emulate the functionality of the others when coupled with software support. The reason to include all three is to ensure highest efficiency in supporting different access patterns.

Figure 17 illustrates this point by comparing performances for applications using remote resources over different channels. For clarity, measurements are normalized to those with the highest performance. The figure shows that different situations benefit from different channels, and none of the channels can be efficiently replaced by another.

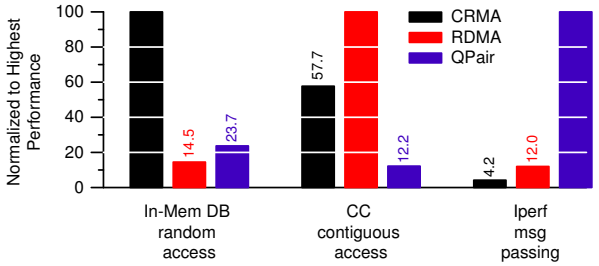


Figure 17: Performance comparison of resource sharing over three different channels

Having three channels optimized for different communication patterns also allows Venice to deliver better performance by using channels collaboratively. Figure 18 shows the impact of using the CRMA channel to facilitate credit-based flow control for communications over the QPair channel. Effective bandwidth of the QPair channel improves from 28% to 51%, depending on packet size. Improvement is greater for small packets.

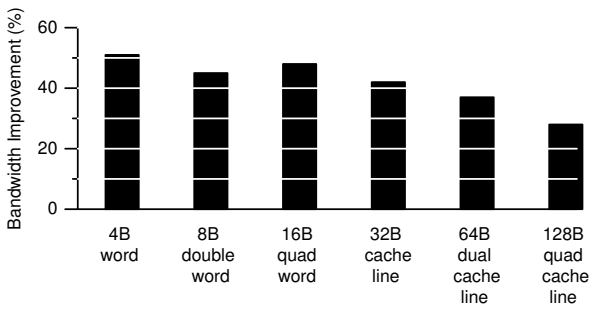


Figure 18: Bandwidth improvement through synergistic operation

7.3 Hardware and programming cost

In a typical implementation, we expect the Venice substrate to support a low-dimension network with a few pins per channel, where each channel delivers around tens of gigabits per second. Our prototype thus includes a custom radix-7 switch plus the three transport layer channels.

Except for the physical layer (PHY), our design is fully synthesizable, and we implement it in Global Foundries 28nm technology with SPG [42], a standard Synopsys ASIC design flow. We use the Synopsis Design Compiler for logic synthesis and the IC Compiler for floorplanning, placement, clock-tree synthesis, and routing. Evaluation results show that the logic can run at 1GHz for the typical corner. Results from EDA tools report a 2.73 mm^2 total layout area and 32KB total SRAM. And we estimate the area of PCIe Gen4 x1 PHY as 0.5 mm^2, which makes the total area of PHYs about 3.5 mm^2. For comparison, note that die sizes of Haswell-EP processors range from 300mm^2 (8 cores) to 600mm^2 (18 cores) at a 22nm technology [43]. The architectural support for Venice thus imposes an insignificant area cost (about 2% of the total chip).

8. CONCLUSIONS AND FUTURE WORK

We propose Venice architectures to break though the hardware boundaries of traditional physical nodes. In these architectures, nodes can easily and efficiently use remote idle resources according to the dynamic workload requirements. Three key techniques enable this: a tightly-coupled inter-node fabric, efficient transport-layer mechanisms for joining resources, and an intelligent runtime to manage resources. Here we focus on the first two layers of hardware support.

From our initial case studies and control experiments, we make the following observations:

1. Our results indicate that resource sharing is a promising feature in data center architectures.

2. Different workloads have different access patterns. For high efficiency, we explore three separate channels to facilitate the access behaviors. We find the three channels to have their own strengths, and it is difficult to substitute one for another without hurting performance. Moreover, the presence of all three channels allows synergistic optimizations in the implementation.

3. Venice architectures can be built at reasonable cost. With further design-space exploration and system optimization, we believe sharing-centric architectures will ultimately be much more cost-effective than statically resourced designs.

Many issues require further exploration to Making resource-sharing robust requires further exploration of many issues, including reliability, scalability, and QoS. Our results point to promising directions for further optimization — for instance, an intelligent runtime is crucial, as distance, topology, and traffic all affect the desirability of co-opting a remote resource. Our maturing prototype serves as a valuable vehicle to support these studies.

9. ACKNOWLEDGMENT

This work was partly supported by the Huawei High Throughput Computing Program for Data Center 3.0 Architecture establishment, and the prototypes in this work are our preliminary efforts to evaluate the key features of Data Center 3.0 [44], such as the optical interconnect and the Pooled Resource Access Protocol (PARP). It was partly supported by the National Science Foundation of China under grant No. 61402439, No. 61402438, and No. 61522212. It was partly supported by the National Science Foundation under grants No. 1217662 and 1514433 and by the Chinese Academy of Sciences President's International Fellowship Initiative under grant No. 2015VTB053. We thank the anonymous reviewers for their valuable comments and suggestions.

10. REFERENCES

[1] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conference on Internet Measurement*, pp. 267–280, Nov. 2010.

[2] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *Proc. 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–18, Mar. 2014.

[3] Mellanox, "Infiniband Performance." http://www.mellanox.com/page/performance_infiniband.

[4] Y. Durand, P. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matus, I. Mavroidis, and J. Thomson, "EUROSERVER: Energy efficient node for European micro-servers," in *Proc. IEEE Euromicro Symposium on Digital System Design*, pp. 206–213, Aug. 2014.

[5] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA highly scalable server," in *Proc. 24th Annual International Symposium on Computer Architecture*, pp. 241–251, June 1997.

[6] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford Dash multiprocessor," *Computer*, vol. 25, pp. 63–79, March 1992.

[7] ScaleMP, "Versatile SMP (vSMP) Architecture." http://www.scalemp.com/technology/versatile-smp-vsmp-architecture/.

[8] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, Nov. 1989.

[9] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: shared memory computing on networks of workstations," *Computer*, vol. 29, pp. 18–28, Feb. 1996.

[10] J. Oleszkiewicz, L. Xiao, and Y. Liu, "Parallel network RAM: effectively utilizing global cluster memory for large data-intensive

parallel programs," in *Proc. International Conference on Parallel Processing*, vol. 1, pp. 353–360, Aug. 2004.

[11] E. Felton and J. Zahorjan, "Issues in the implementation of a remote memory paging system," Tech. Rep. 91-03-09, University of Washington, Department of Computer Science and Engineering, Mar. 1991.

[12] M. Hines, M. Lewandowski, and K. Gopalan, "Anemone: Adaptive network memory engine," in *Proc. 20th ACM Symposium on Operating Systems Principles*, p. 1, Oct. 2005.

[13] H. Jin, X.-H. Sun, Y. Chen, and T. Ke, "REMEM: Remote memory as checkpointing storage," in *Proc. 2nd International Conference on Cloud Computing Technology and Science*, pp. 319–326, Dec. 2010.

[14] M. Feeley, W. Morgan, E. Pighin, A. Karlin, H. Levy, and C. Thekkath, "Implementing global memory management in a workstation cluster," in *Proc. 15th ACM Symposium on Operating Systems Principles*, pp. 201–212, Dec. 1995.

[15] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proc. 1st USENIX Conference on Operating Systems Design and Implementation*, p. 19, Nov. 1994.

[16] E. Anderson, J. Brooks, C. Grassl, and S. Scott, "Performance of the Cray T3E multiprocessor," in *Proc. ACM/IEEE 1997 Conference on Supercomputing*, pp. 39–39, Nov. 1997.

[17] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, "The PERCS high-performance interconnect," in *Proc. IEEE 18th Annual Symposium on High Performance Interconnects*, pp. 75–82, 2010.

[18] Y. Ajima, Y. Takagi, T. Inoue, S. Hiramoto, and T. Shimizu, "The Tofu Interconnect," in *Proc. IEEE 19th Annual Symposium on High Performance Interconnects*, pp. 87–94, Aug. 2011.

[19] M. Xie, Y. Lu, K. Wang, L. Liu, H. Cao, and X. Yang, "Tianhe-1A interconnect and message-passing services," *IEEE Micro*, vol. 32, pp. 8–20, Jan. 2012.

[20] A. Rao, "AMD | SeaMicro Technology Overview." http://www.seamicro.com/sites/default/files/SM_TO01_64_v2.7.pdf, Oct. 2012.

[21] T. P. Morgan, "On-Chip Networking May Survive Calxeda Shutdown." http://www.enterprisetech.com/2014/01/02/chip-networking-may-survive-calxeda-shutdown, Feb. 2014.

[22] M. Kumar, "Rack scale architecture for cloud," in *Intel IDF*, 2013.

[23] D. Mayhew and V. Krishnan, "PCI Express and advanced switching: Evolutionary path to building next generation interconnects," in *Proc. 11th Symposium on High Performance Interconnects*, pp. 21–29, Aug. 2003.

[24] R. Hou, T. Jiang, L. Zhang, P. Qi, J. Dong, H. Wang, X. Gu, and S. Zhang, "Cost effective data center servers," in *Proc. IEEE International Symposium on High Performance Computer Architecture*, pp. 179–187, Feb. 2013.

[25] J. Regula, "Integrating rack level connectivity into a PCI Express switch," in *Proc. Hot Chips: A Symposium on High Performance Chips*, pp. 259–266, Aug. 2013.

[26] K. Lim, Y. Turner, J. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *Proc. 18th International Symposium on High Performance Computer Architecture*, pp. 1–12, Feb. 2012.

[27] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *Proc. 35th Annual International Symposium on Computer Architecture*, pp. 315–326, June 2008.

[28] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. Reinhardt, and T. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proc. 36th Annual International Symposium on Computer Architecture*, pp. 267–278, June 2009.

[29] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, G. Jan, G. Michael, H. S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Yi, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. 41st Annual International Symposium on Computer Architecuture*, pp. 13–24, June 2014.

[30] R. Noronha and D. Panda, "Designing high performance DSM systems using InfiniBand features," in *Proc. 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 467–474, Apr. 2004.

[31] "Zynq-7000 All Programmable SoC." White Paper, 2014.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 319–326, Oct. 2002.

[33] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107–113, Jan. 2008.

[34] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conference on Hot Topics in Cloud Computing*, p. 10, June 2012.

[35] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: a big data benchmark suite from internet services," in *Proc. 20th IEEE International Symposium On High Performance Computer Architecture (HPCA)*, pp. 488–499, Feb. 2014.

[36] Graph500, "http://www.graph500.org/."

[37] Oracle Berkeley DB, "http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html."

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd International Symposium on Computer Architecture*, pp. 24–36, May 1995.

[39] Iperf, "http://iperf.fr/."

[40] Oracle Corp., "MySQL: The world's most popular open-source database." http://www.mysql.com, 2014.

[41] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Magazine*, Aug. 2009.

[42] Synopsys. `http://www.synopsys.com/Community/Partners/CommonPlatform/Pages/ReferenceFlow.aspx`.

[43] Wiki, "Intel Xeon microprocessors, http://en.wikipedia.org/wiki/list_of_intel_xeon_microprocessors#haswell-based_xeons."

[44] Huawei. http://www.huawei.com/en/industry-insights/huawei-voices/white-papers.